Factor Graphs and the Sum-Product Algorithm

Frank R. Kschischang

Department of Electrical & Computer Engineering
University of Toronto

(Joint work with Brendan Frey and Hans-Andrea Loeliger.)

IMA Summer Program on Codes, Systems, and Graphical Models
August 2, 1999

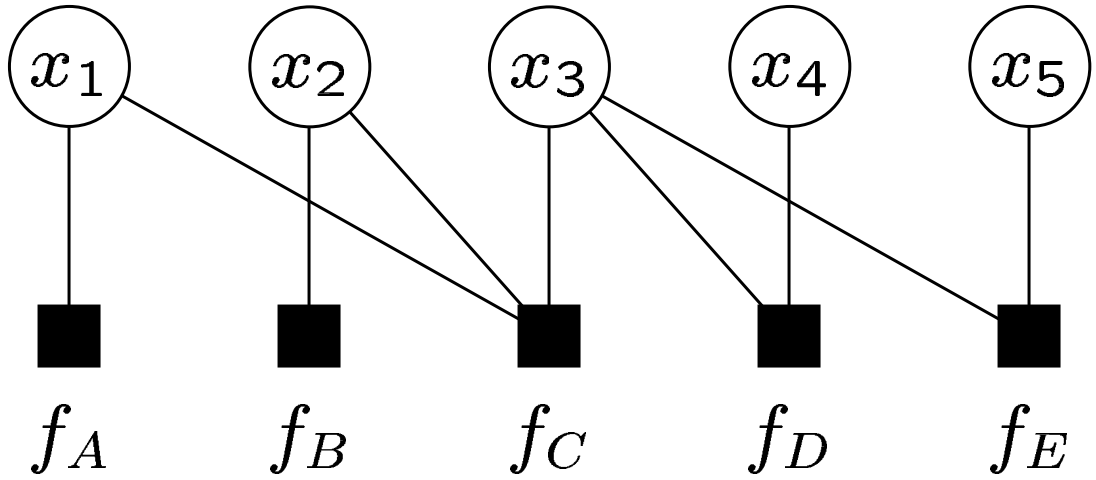
Many algorithms developed in computer science and engineering deal with complicated "global" functions of many variables.

They often derive computational efficiency by exploiting the way in which the global function *factors* into a product of simpler "local" functions.

Such a factorization can be visualized using a **factor graph**.

Example:

$$g(x_1, x_2, x_3, x_4, x_5) = f_A(x_1) f_B(x_2) f_C(x_1, x_2, x_3) f_D(x_3, x_4) f_E(x_3, x_5).$$



Factor graphs are *bipartite*; they represent the "is an argument of" relation between variables and local functions.

As we will see, the structure of a factor graph

1. not only encodes the factorization of the global function; but
2. when cycle-free, encodes an algorithm, **the sum-product algorithm**, for computing marginal functions.

A wide class of algorithms can be captured as specific instances of the sum-product algorithm operating in an appropriate factor graph.

Factor Graph Genealogy

Factor graphs are descendents of bipartite graph representations for error-correcting codes:

- Tanner graphs (1981)
- Tanner/Wiberg/Loeliger/Kötter (TWLK) graphs (1995)

Sum-product and min-sum algorithms were developed by these authors (and even earlier by Gallager) for decoding.

*A*HA!

Aji and McEliece recognize that many algorithms—even some non-probabilistic ones—solve the "marginalize product of functions" (MPF) problem.

(Presented at Ulm ISIT, 1997, GDL paper, 1998).

At same ISIT, corridor discusions among "Ulm group" leads to notion of factor graph.

Factor graphs are also closely related to **graphical models** for probability distributions, e.g.,

- Markov random fields,
- Bayesian networks;

since such graphical models also attempt to capture a factorization: in this case of the joint probability mass function of a collection of random variables.

Part I

The Sum-Product Algorithm

Assume that a variable x_i takes on values in a finite *domain*.

For now, we can assume all functions are real-valued, i.e., the codomain of all functions is $\mathbf{R}$. (It is possible to generalize both the codomain, and even the notion of "product.")

As a working example, let the global function be the conditional probability mass function for a collection of discrete random variables, given the observation of a related quantity.

(E.g., the conditional joint pmf for codeword symbols given the observed channel output).

We introduce an eccentric summation notation: the **not-sum**.

Instead of indicating the variables being summed over, indicate those *not* being summed over.

Thus, if h is a function of three variables x_1, x_2, and x_3, the "not-sum over x_2" is denoted

$$\sum_{\sim\{x_2\}} h(x_1, x_2, x_3) := \sum_{x_1, x_3} h(x_1, x_2, x_3).$$

Implicit is the assumption that each variable being summed, is summed over all values of its domain.

In terms of the not-sum, it is easy to define *marginal functions* associated with a global function.

Given $g(x_1, \ldots, x_n)$, define, for $i = 1, \ldots, n$,

$$g_i(x_i) := \sum_{\sim\{x_i\}} g(x_1, \ldots, x_n).$$

The term "marginal function" derives from the corresponding concept in probability theory.

Suppose, for example,

$$g(x_1, \ldots, x_5) = f_A(x_1) f_B(x_2) f_C(x_1, x_2, x_3) f_D(x_3, x_4) f_E(x_3, x_5).$$

and we wish to compute

$$g_1(x_1) := \sum_{\sim\{x_1\}} g(x_1, \ldots, x_5).$$

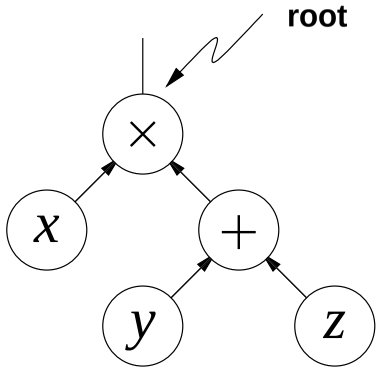
In conventional summation notation, using the distributive law, we might write:

$$g_1(x_1) = f_A(x_1) \sum_{x_2,x_3} \left(f_B(x_2) f_C(x_1,x_2,x_3) \left(\sum_{x_4} f_D(x_3,x_4) \right) \left(\sum_{x_5} f_E(x_3,x_5) \right) \right).$$

or, in not-sum notation,

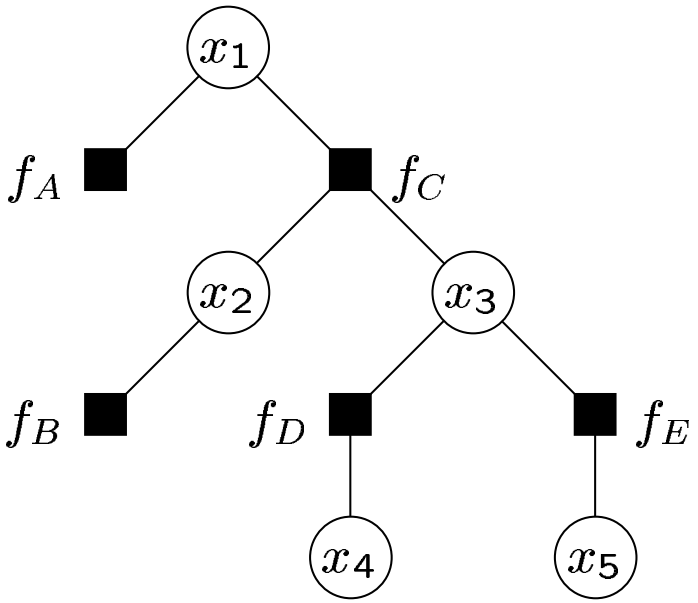
$$g_1(x_1) = f_A(x_1) \sum_{\sim\{x_1\}} \left(f_B(x_2) f_C(x_1,x_2,x_3) \left(\sum_{\sim\{x_3\}} f_D(x_3,x_4) \right) \left(\sum_{\sim\{x_3\}} f_E(x_3,x_5) \right) \right).$$

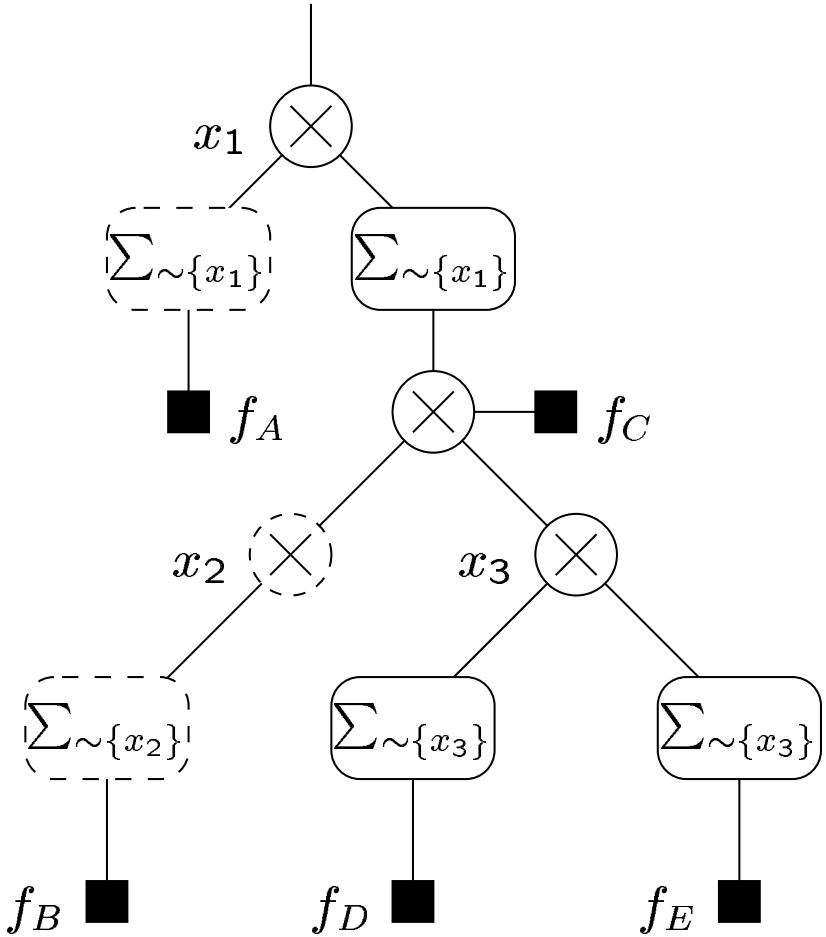
Expression trees:



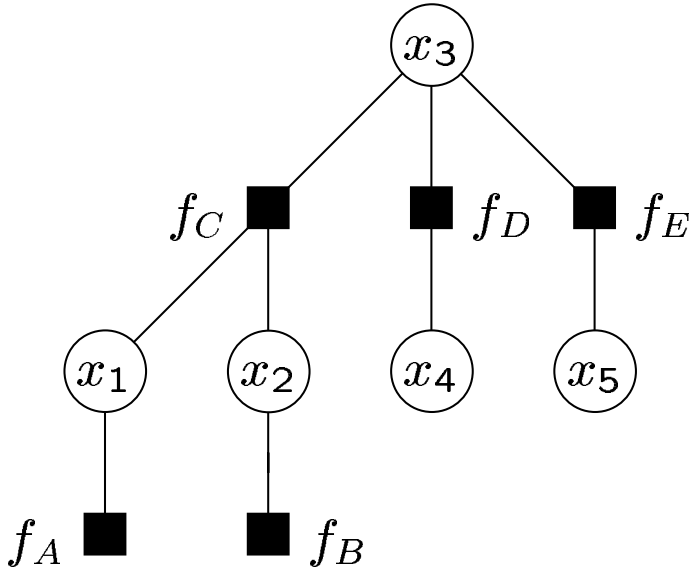
An expression tree for $x \times (y + z)$.

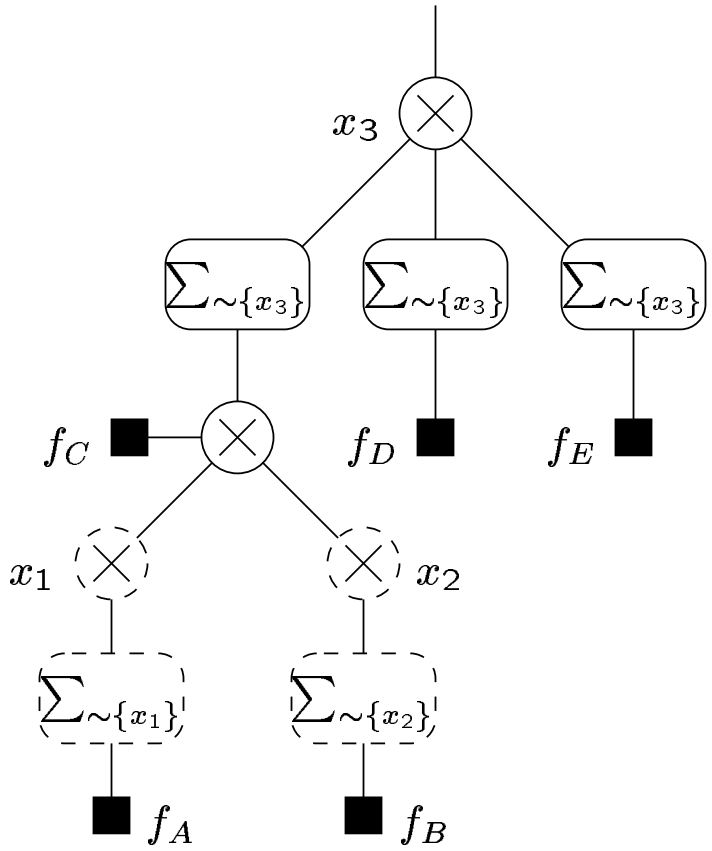
$$f_A(x_1) \times \sum_{\sim\{x_1\}} \left(f_B(x_2) \times f_C(x_1, x_2, x_3) \times \left(\sum_{\sim\{x_3\}} f_D(x_3, x_4) \right) \times \left(\sum_{\sim\{x_3\}} f_E(x_3, x_5) \right) \right)$$



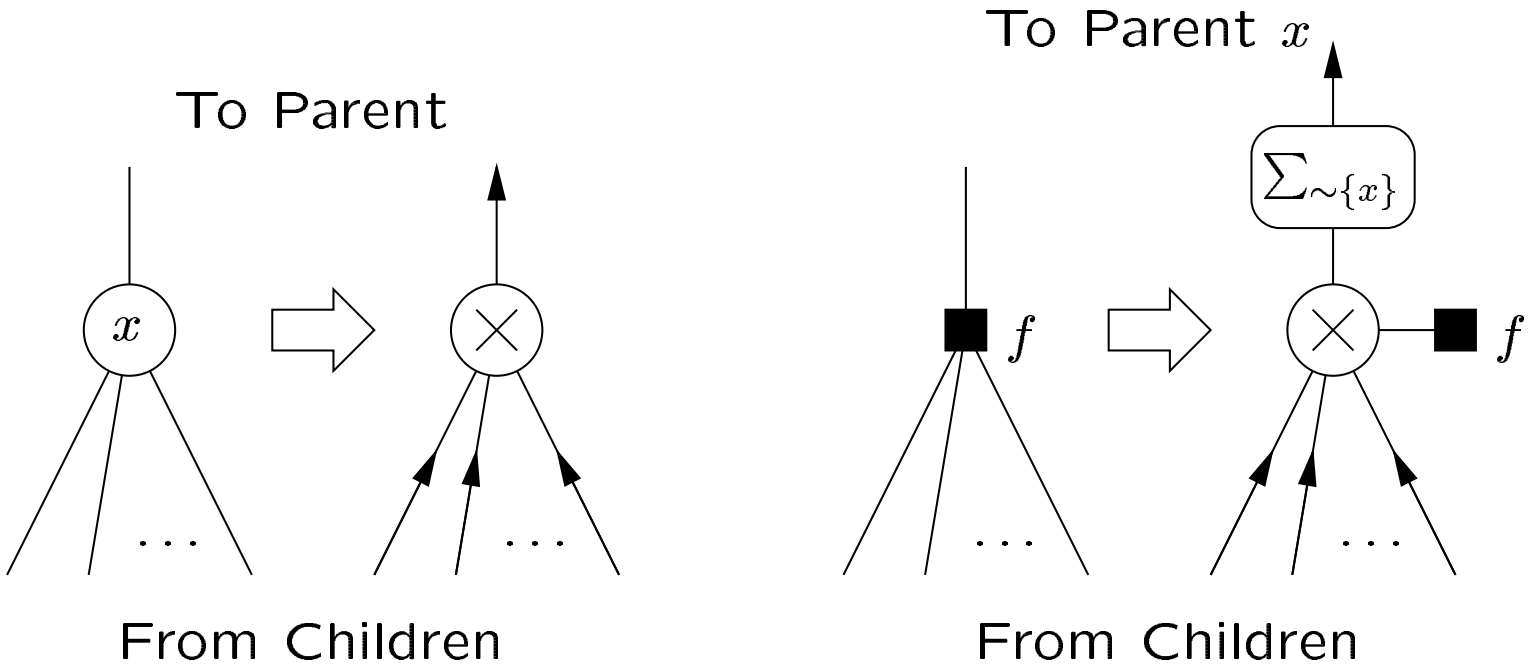


$$g_3(x_3) = \left(\sum_{\sim\{x_3\}} f_A(x_1) f_B(x_2) f_C(x_1, x_2, x_3) \right) \times \left(\sum_{\sim\{x_3\}} f_D(x_3, x_4) \right) \times \left(\sum_{\sim\{x_3\}} f_E(x_3, x_5) \right)$$





Local transformation, converting a rooted cycle-free factor graph to an expression tree:



We see that when the factor graph is a tree, by making x_i the *root*, the factor graph encodes an expression, and therefore an **algorithm** for computing $g_i(x_i)$.

The algorithm is simple:

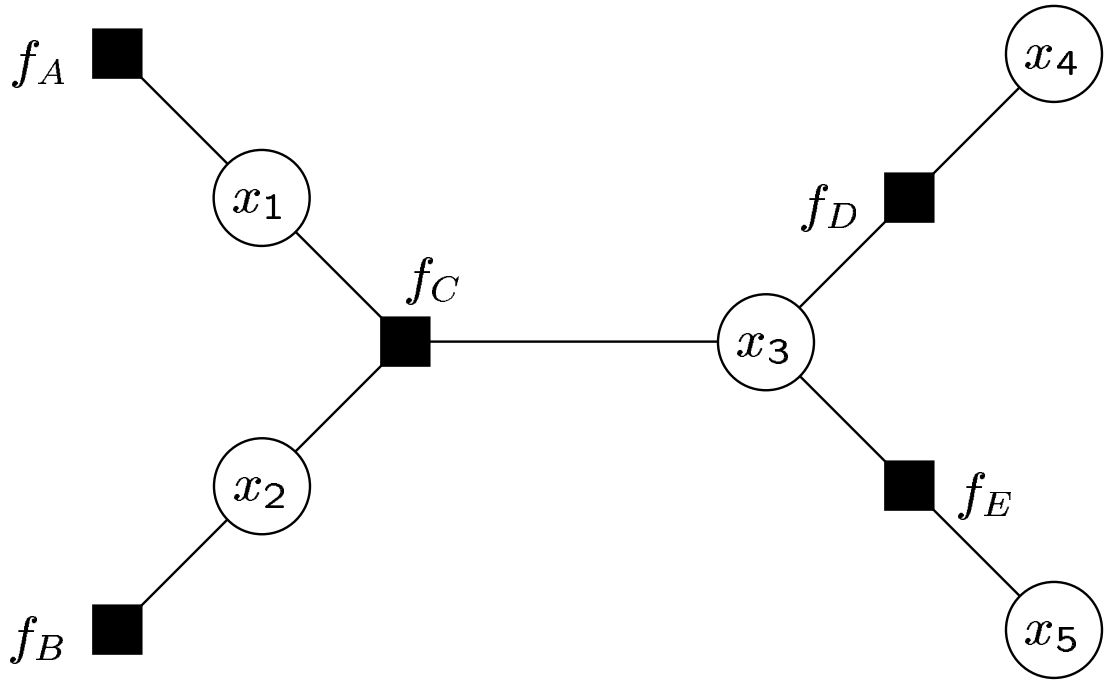
1. (product rule) At a variable node x_i, take the product of expressions formed at the descendants of x_i.

2. (sum-product rule) At a function node f, take the product of f with expressions formed at the descendants of f; then perform the not-sum over the parent of f.

We compute sums and products $\therefore$ call it the *sum-product* algorithm.

The sum-product algorithm can be described very naturally as a *message-passing* algorithm:

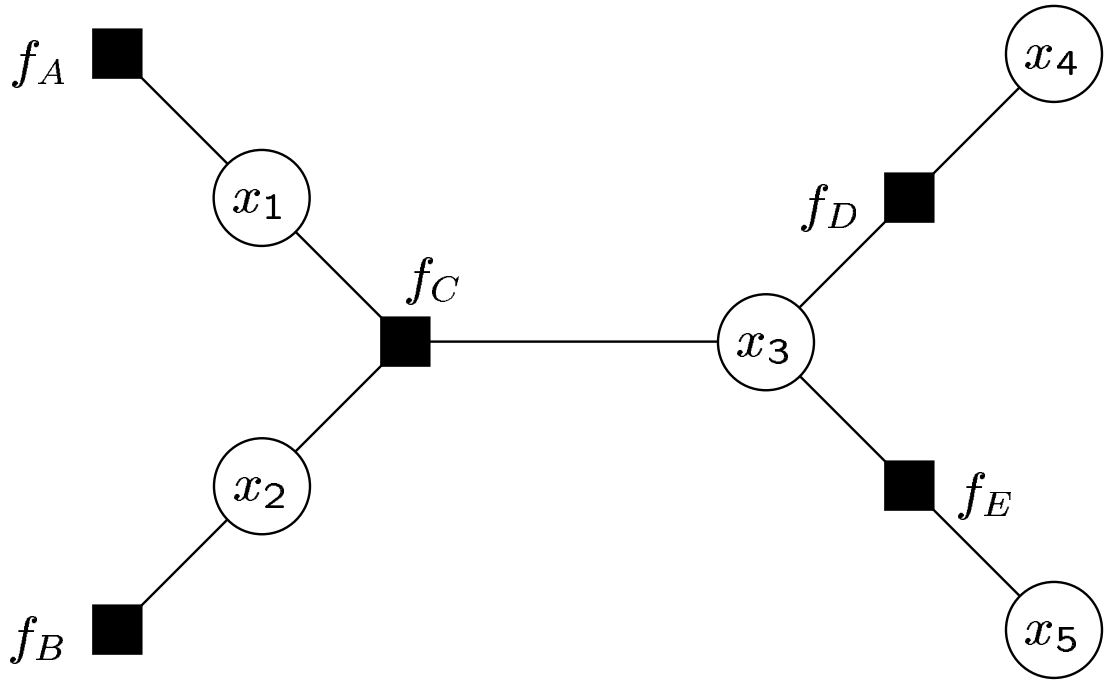
- Vertices are "processors."
- Edges are "channels" between processors.
- "Messages" sent over channels are simply appropriate descriptions of local functions.
- Any particular processor can "fire" once it has received messages from its children.
- Start from the leaves and send messages "up" towards the root.
- The marginal function $g_i(x_i)$ is the "final" message; namely, the product of all messages sent to x_i.

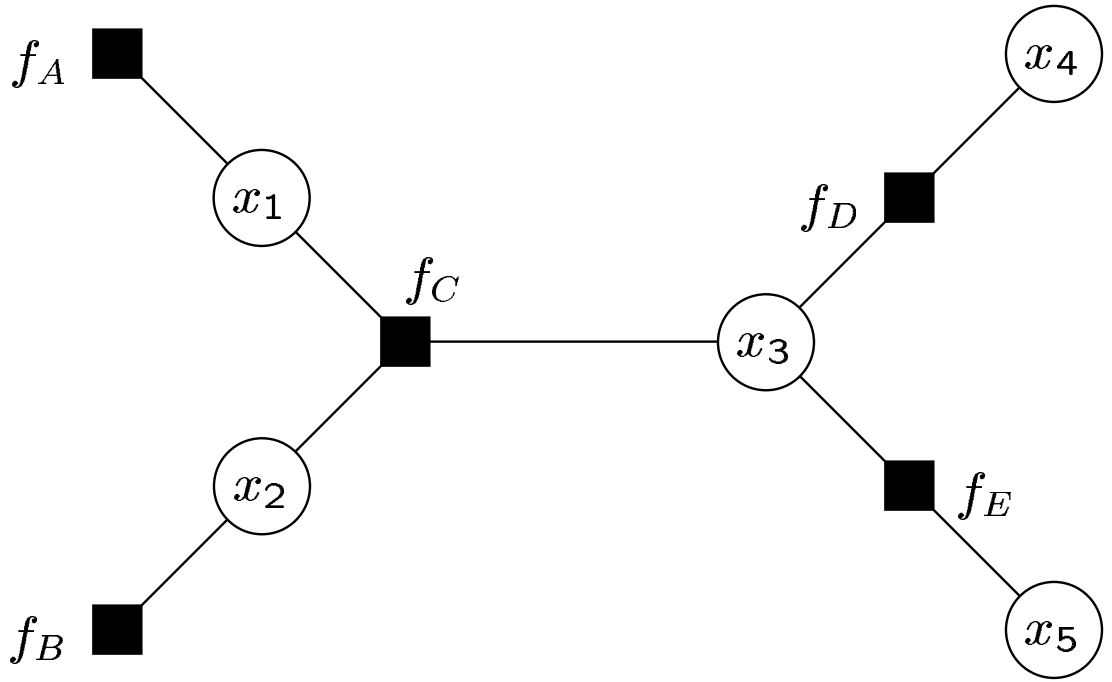

f_A
x_1
f_C
x_2
f_B
x_3
f_D
x_4
f_E
x_5

Often we are interested in computing $g_i(x_i)$ for all i.

We can do this by *overlaying* multiple copies of the single-i algorithm on a single graph.

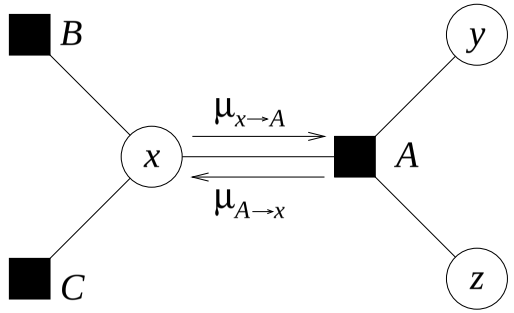
Implicitly *every* variable of the tree may at some point be regarded as root.


f_A
x_1
f_C
x_2
f_B
x_3
x_4
f_D
f_E
x_5


f_A
x_1
f_C
x_2
f_B
x_3
x_4
f_D
f_E
x_5

The sum-product algorithm in a tree:

- A processor "fires" when it has received messages from all but one of its neighbors.
- Again, the process starts at the leaf vertices, which "fire" immediately.
- The one remaining neighbor is the "temporary parent;" the others are the "children."
- When a message is received from the temporary parent, each "child" is, in turn, regarded as the parent.
- The process terminates when all vertices have received messages from all neighbors.
- Again, $g_i(x_i)$ is the product of all messages received by x_i.



variable to function: (the product rule)

$$\begin{aligned}\mu_{x\to A}(x) &:= \prod_{F\in n(x)\setminus\{A\}} \mu_{F\to x}(x) \\ &= \mu_{B\to x}(x)\cdot\mu_{C\to x}(x).\end{aligned}$$

function to variable: (the sum-product rule)

$$\begin{aligned}\mu_{A\to x}(x) &:= \sum_{\sim\{x\}} f_A(x,x_1,\ldots,x_J)\cdot\prod_{i=1}^{J}\mu_{x_i\to A}(x_i) \\ &= \sum_{\sim\{x\}} f_A(x,y,z)\cdot\mu_{y\to A}(y)\cdot\mu_{z\to A}(z).\end{aligned}$$

Notes:

- Exactly one message passes in each direction over any edge, i.e., the sum-product algorithm in a tree is a "two-way" algorithm.
- In a finite tree with E edges, the algorithm terminates after $2E$ messages have been passed, i.e., in a finite number of steps.
- The message that passes (in either direction) over an edge $\{x, f\}$ is always a function of x, i.e., a single-argument function.
- At termination, $g_i(x_i) = \mu_{x_i \to f}(x_i) \times \mu_{f \to x_i}(x_i)$ for any neighbor f of x_i.

We can generalize the notion of "sum" to "summary:"

Essentially, a "marginal" function $f_i(x_i)$ of a function $f(x_1, \ldots, x_n)$, can, for each value of x_i be regarded as a "summary" of the multiple function values that f assumes as the variables *other* than x_i range over their respective domains.

E.g., if $f(x_1, x_2, x_3)$ is a function of three binary variables,

- $f_1(0)$ is a summary of $f(0,0,0)$, $f(0,0,1)$, $f(0,1,0)$, $f(0,1,1)$,
- $f_1(1)$ is a summary of $f(1,0,0)$, $f(1,0,1)$, $f(1,1,0)$, $f(1,1,1)$.

In general a variety of summary operators are possible.

We can obtain a summary operator from a binary operation in an abelian semigroup (see the Aji/McEliece GDL paper). Designate the operation by '+'.

e.g.,

- $+$ is real-valued summation: Then $F_x(x)$ is the *marginal* function.
- $+$ is "max" or "min": Then $F_x(x)$ is the *maximum* (resp., *minimum*) of $F(x, y, z, \ldots)$ taken over all argument configurations with fixed x.

We assume the distributive law:

$$x(y+z) = xy + xz$$

which makes the domain R of F a semiring.

(See Aji/McEliece, "The Generalized Distributive Law" for more!)

In terms of general summary operators:

Sum-product algorithm (in words)

The message sent by a node on an edge is the product of the messages received on all *other* edges with the local function at the node, summarized for the variable associated with the edge.

Part II

Codes and Probability Distributions

While many types of functions might potentially be represented by factor graphs, two particular classes of functions stand out: **characteristic functions** for sets and **probability distributions**. These can often be viewed as models — set theoretic and probabilistic, resp., — of a physical system.

Characteristic Functions:

Suppose the global function g is a binary indicator function, i.e.,

$$g(w) = [w \in B] = \begin{cases} 1 & \text{if } w \in B, \\ 0 & \text{otherwise.} \end{cases}$$

where $B \subset W$ is a subset of the possible configurations: the set of *valid behaviors* or *codewords*.

"Iverson's convention" (from APL): if P is a Boolean proposition, then

$$[P] = \begin{cases} 1 & \text{if P;} \\ 0 & \text{otherwise.} \end{cases}$$

Often, can express membership in B by a series of "local checks." For example, every linear code block code C over F_q can be specified by a *parity check matrix* H. Given H then $C = \{v : vH^T = 0\}$.

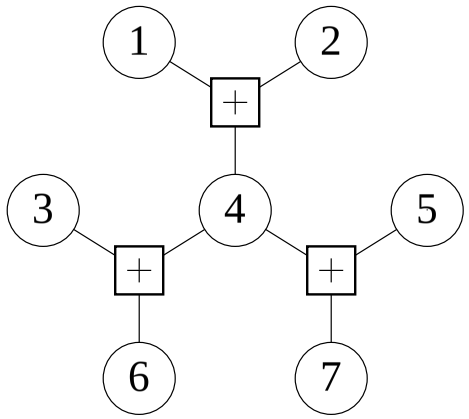
For example, over F_2, take

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

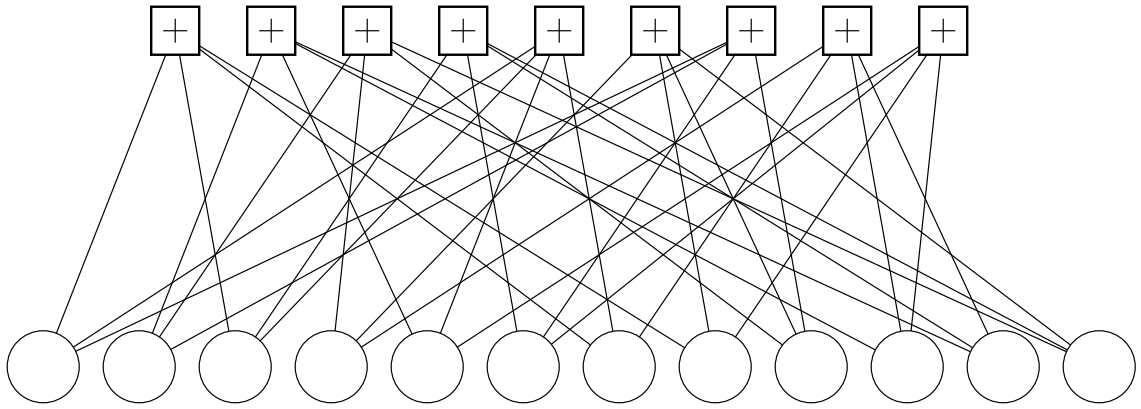
$$\mathbf{v}H^T = (v_1, \ldots, v_7)H^T = 0 \rightarrow \begin{cases} v_1 \oplus v_2 \oplus v_4 = 0 \\ v_3 \oplus v_4 \oplus v_6 = 0 \\ v_4 \oplus v_5 \oplus v_7 = 0 \end{cases} ; \text{ therefore,}$$

$$[\mathbf{v} \in C] = [v_1 \oplus v_2 \oplus v_4 = 0] \cdot [v_3 \oplus v_4 \oplus v_6 = 0] \cdot [v_4 \oplus v_5 \oplus v_7 = 0].$$

$$[\mathbf{v} \in C] = [v_1 \oplus v_2 \oplus v_4 = 0] \cdot [v_3 \oplus v_4 \oplus v_6 = 0] \cdot [v_4 \oplus v_5 \oplus v_7 = 0].$$



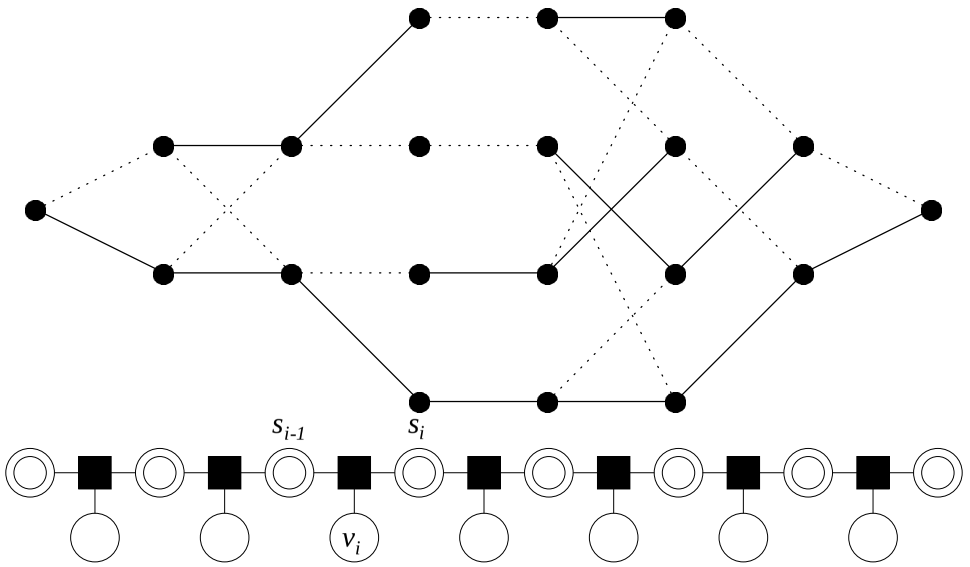
Low-Density Parity-Check Codes (R. G. Gallager, 1962)



A (3,4) LDPC code.

Recent irregular enhancements of LDPC codes give rise to the most powerful long, yet decodable, codes known (Richardson, Shokrollahi, Urbanke, 1999).

Factor Graph for a Trellis



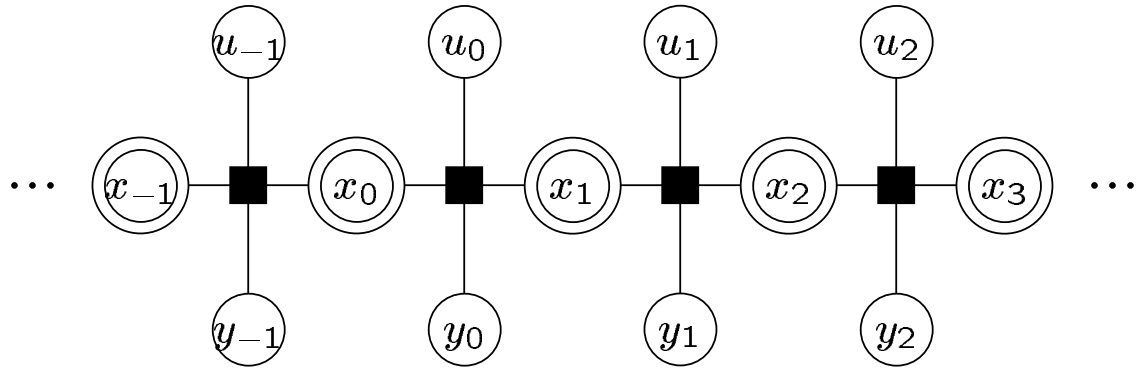
T_i: set of edges in the i trellis section

$$f(v_1, \ldots, v_7, s_0, \ldots, s_7) = \prod_{i=1}^{7} [(s_{i-1}, v_i, s_i) \in T_i]$$

Classical linear state-space model:

$$\begin{aligned} x(j+1) &= Ax(j) + Bu(j) \\ y(j) &= Cx(j) + Du(j) \end{aligned}$$

$$x \in F^m, u \in F^k, y \in F^n$$

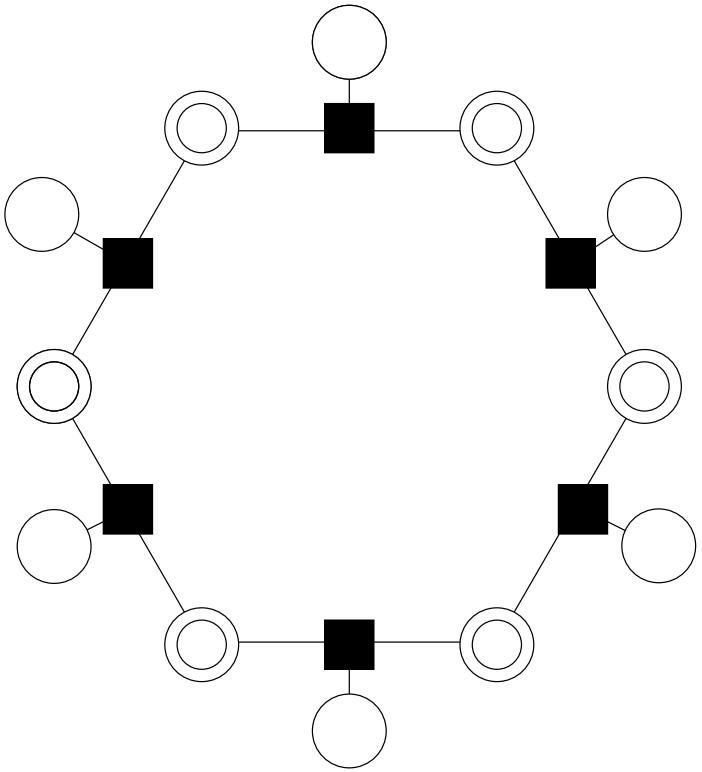


The time-j check function $f(x(j), u(j), y(j), x(j+1))$: $F^m \times F^k \times F^n \times F^m \to \{0, 1\}$ is

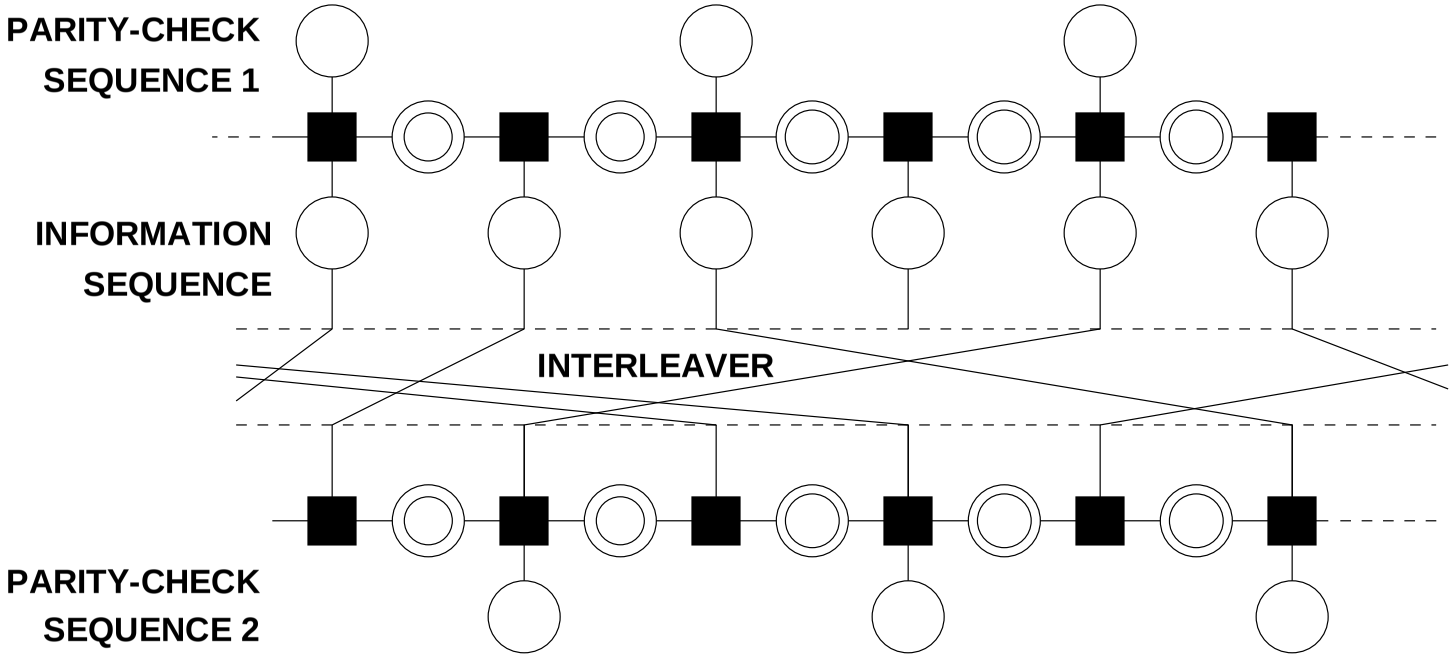
$$f(x(j), u(j), y(j), x(j+1)) =$$

$$[x(j+1) = Ax(j) + Bu(j)][y(j) = Cx(j) + Du(j)].$$

Factor Graph for a Tail-Biting Trellis



Factor graph for a turbo code



(N. Wiberg, Ph.D. dissertation, 1996.)

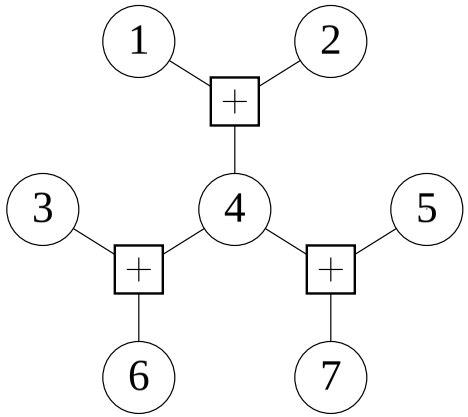
A Posteriori Probabilities

Suppose we transmit $(x_1, \ldots, x_N)$ over a memoryless channel, and observe $(y_1, \ldots, y_N)$.

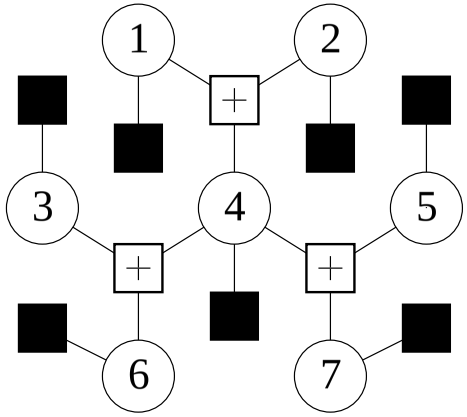
The *a posteriori* joint probability distribution of $\{x_1, \ldots, x_N\}$ is then *proportional* to

$$f(x_1, x_2, \ldots, x_n) = \prod_{E \in Q} f_E(x_E) \prod_{i=1}^{n} f(y_i|x_i),$$

The factor graph for the function indicating code membership is simply augmented with $f(y_i|x_i)$, the channel likelihood functions.



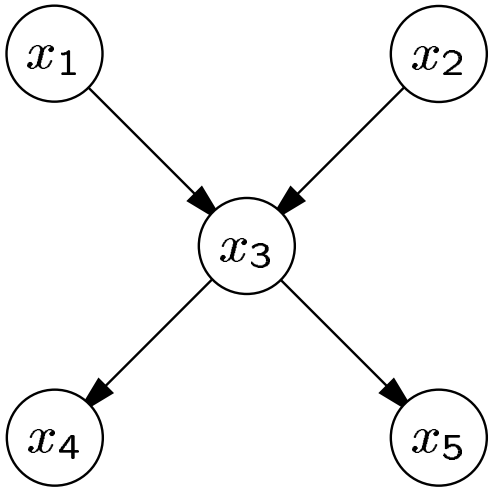
a priori distribution
(uniform)

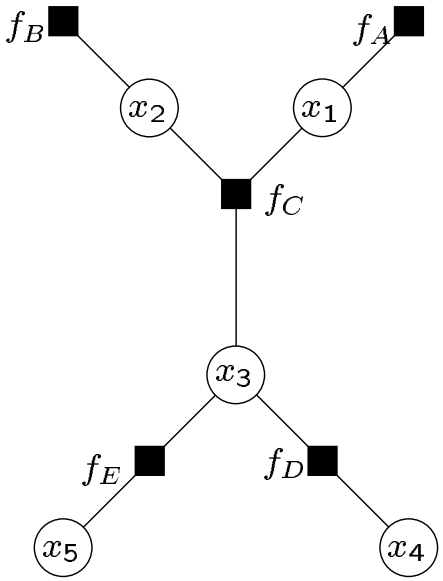


a posteriori distribution
(up to scale factor)

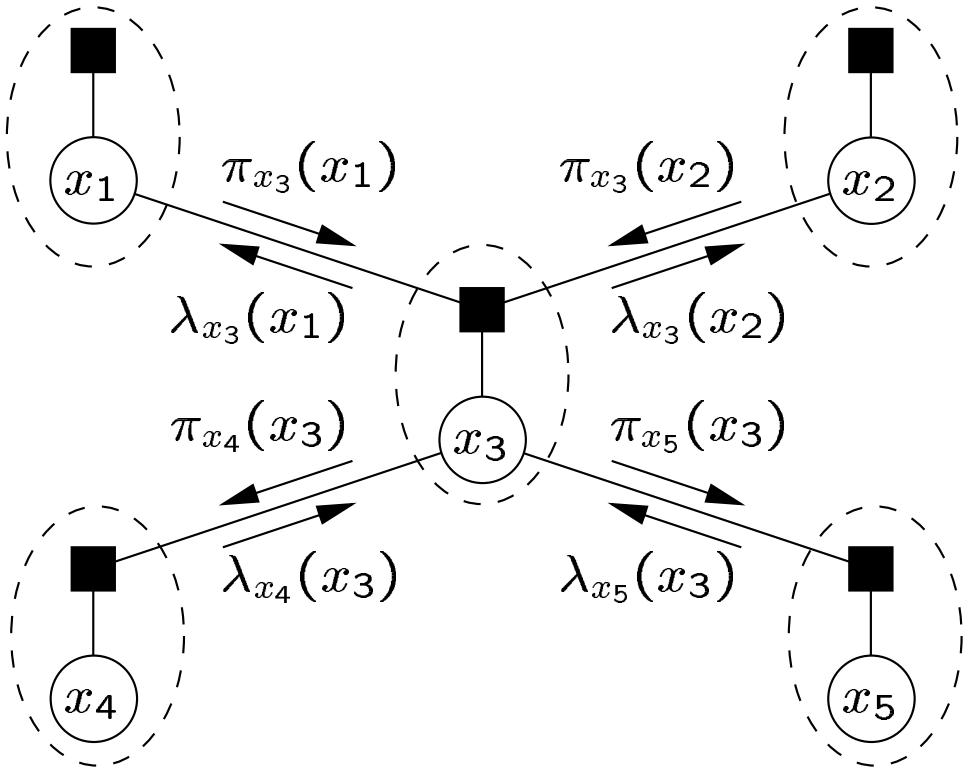
Memoryless channel $\rightarrow$ augment code's factor graph with "Dongles"

Bayesian Networks





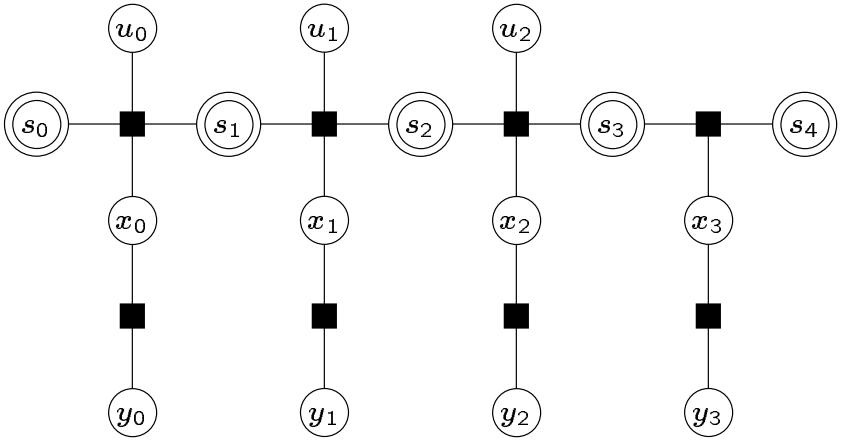
$$f(x_1, \ldots, x_5) = f_A(x_1) f_B(x_2) f_C(x_3|x_1, x_2) f_D(x_4|x_3) f_E(x_5|x_3)$$

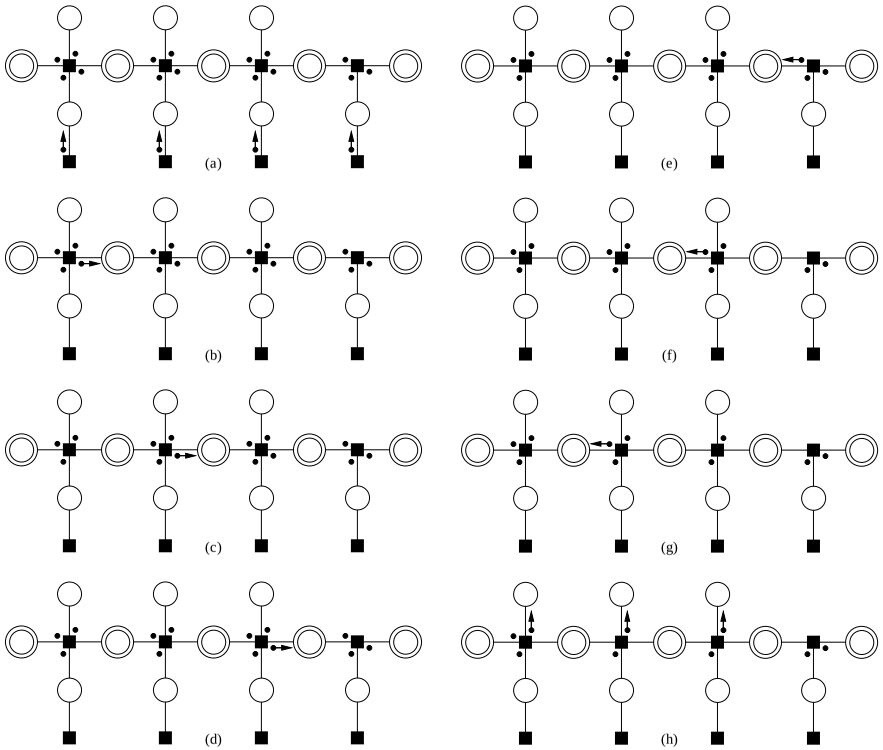


Belief propagation as an instance of the sum-product algorithm.

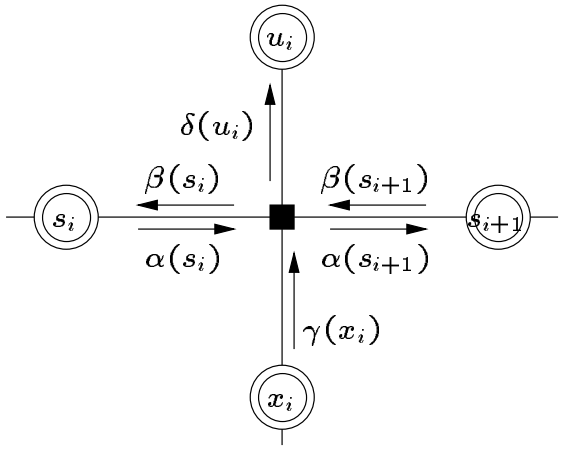
Applications

Hidden Markov Model:





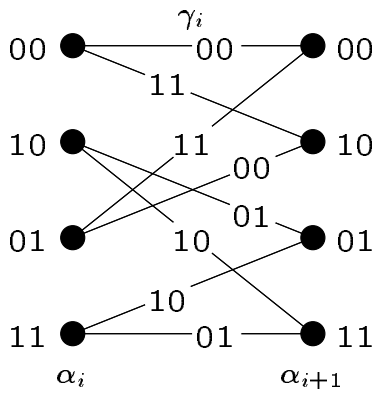
The forward/backward BCJR algorithm.



$$\alpha(s_{i+1}) = \sum_{s_i} \sum_{x_i} [(s_i, x_i, s_{i+1}) \in T_i] \alpha(s_i) \gamma(x_i)$$

$$\beta(s_i) = \sum_{s_{i+1}} \sum_{x_i} [(s_i, x_i, s_{i+1}) \in T_i] \beta(s_{i+1}) \gamma(x_i)$$

$$\delta(u_i) = \sum_{s_i} \sum_{s_{i+1}} \sum_{x_i} [(s_i, x_i, s_{i+1}) \in T_i] \alpha(s_i) \beta(s_{i+1}) \gamma(x_i)$$



Can express as sums over trellis edges:

$$\alpha(s_{i+1}) = \sum_{e \in E_i(s_{i+1})} \alpha(e)\gamma(e)$$

$$\beta(s_i) = \sum_{e \in E_i(s_i)} \beta(e)\gamma(e).$$

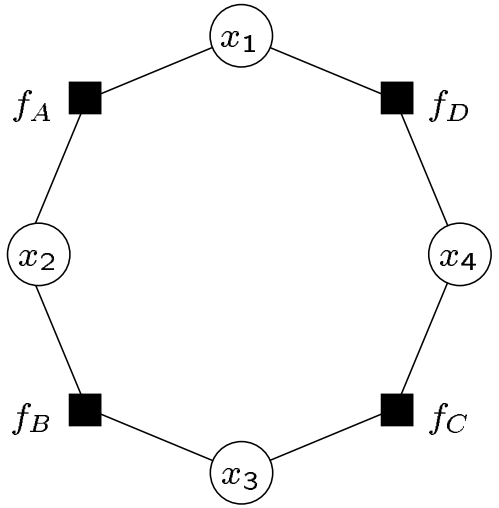
In the min-sum semiring $\rightarrow$ min-sum algorithm. One-way version with traceback $\rightarrow$ Viterbi algorithm. (Two-way version gives the same result.)

Gaussian variables + summarization by integration $\rightarrow$ Kalman filtering.

Coping with Cycles:

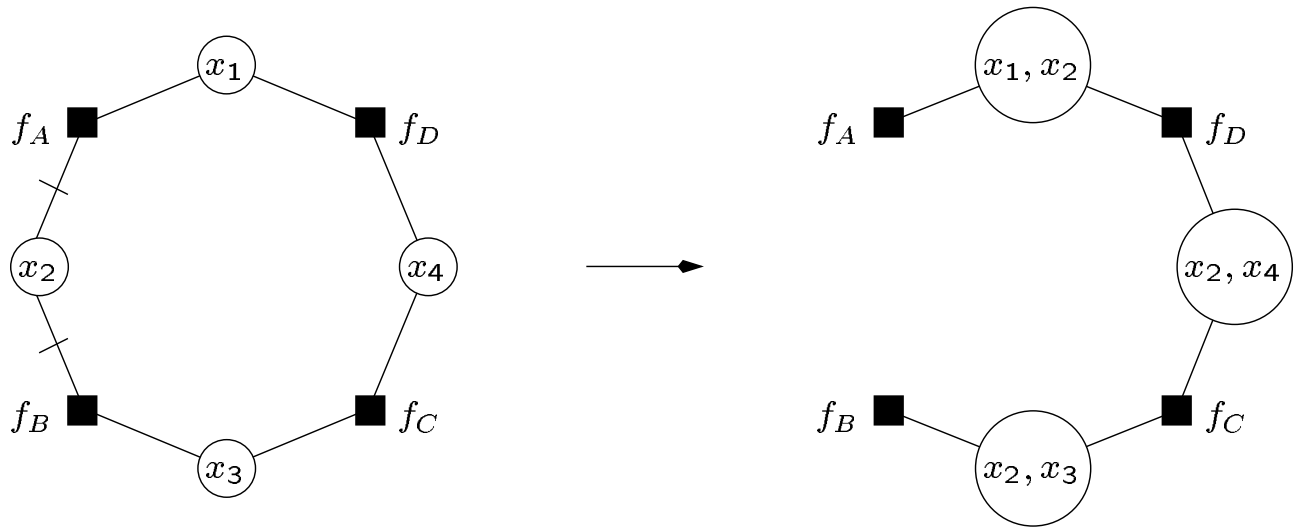
What happens when the factor graph is *not* cycle-free? For example, suppose we have

$$g(x_1, x_2, x_3, x_4) = f_A(x_1, x_2) f_B(x_2, x_3) f_C(x_3, x_4) f_D(x_1, x_4).$$



$$
\begin{aligned}
g_1(x_1) &:= \sum_{\sim\{x_1\}} g(x_1, x_2, x_3, x_4) \\
&= \sum_{x_2, x_3, x_4} f_A(x_1, x_2) f_B(x_2, x_3) f_C(x_3, x_4) f_D(x_1, x_4) \\
&= \sum_{x_2} f_A(x_1, x_2) \sum_{x_4} f_D(x_1, x_4) \sum_{x_3} f_B(x_2, x_3) f_C(x_3, x_4) \\
&= \sum_{\sim\{x_1\}} f_A(x_1, x_2) \sum_{\sim\{x_1, x_2\}} f_D(x_1, x_4) \sum_{\sim\{x_2, x_4\}} f_B(x_2, x_3) f_C(x_3, x_4)
\end{aligned}
$$

Note that the "not-sums" now involve more than one variable.



Equivalent to forming a spanning tree by cutting the loop, with x_2 "carried" around the cycle.

$\rightarrow$ Junction Tree Property (see Aji and McEliece)

(A variable x_i must be "carried" over all regions of the sub-spanning-tree in which x_i is involved.)

The FFT (We follow Aji and McEliece, 1997)

The N-point discrete Fourier transform:

$$F[k] = \sum_{n=0}^{N-1} f[n] W_N^{nk}$$

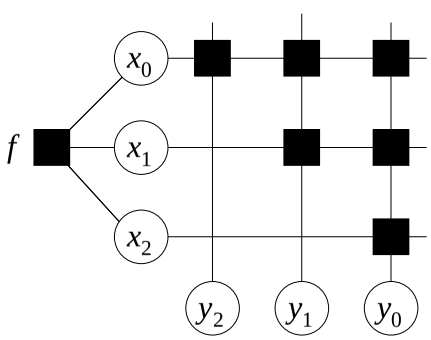
where $W_N = \exp(-j2\pi/N)$.

Take $N = 8$ and $x_0, x_1, x_2, y_0, y_1, y_2 \in \{0, 1\}$.

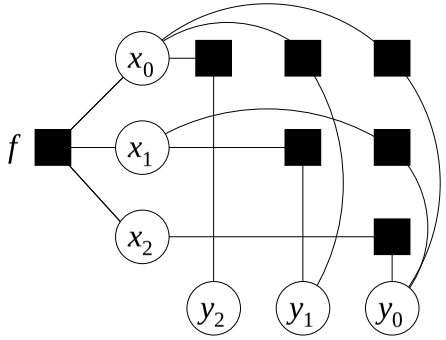
Write $n = 4x_2 + 2x_1 + x_0$, $k = 4y_2 + 2y_1 + y_0$.

DFT kernel function $\Rightarrow g(x_0, x_1, x_2, y_0, y_1, y_2)$

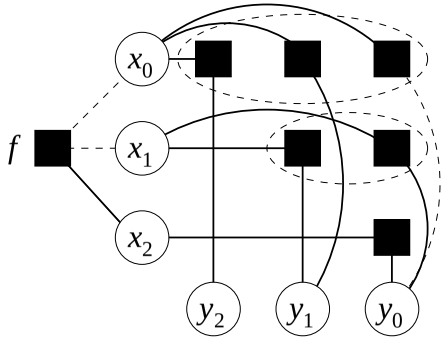
$$\begin{aligned} &= f[4x_2 + 2x_1 + x_0] W^{(4x_2+2x_1+x_0)(4y_2+2y_1+y_0)} \\ &= f[4x_2 + 2x_1 + x_0] \times \\ &\quad (-1)^{x_2 y_0} (-1)^{x_1 y_1} (-1)^{x_0 y_2} (-j)^{x_0 y_1} (-j)^{x_1 y_0} W^{x_0 y_0} \end{aligned}$$



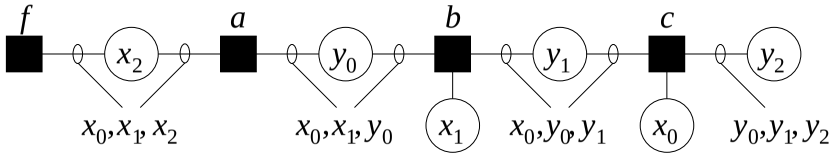
Factor Graph



Factor Graph



A Spanning Tree



An FFT

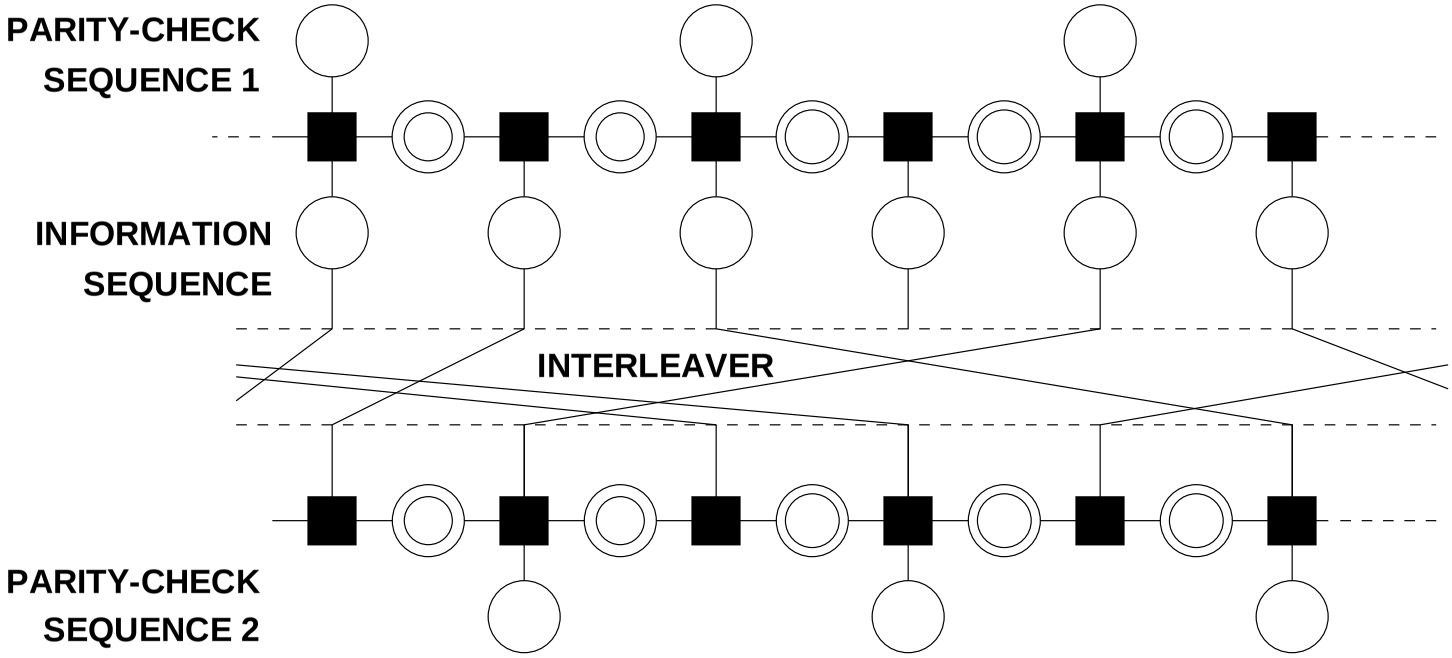
Coping with Cycles (Part II):

Ignore them and hope for the best!

This is the approach favored in decoding, since exact marginalization is computationally infeasible, and since the approximate APPs are quantized in the end.

This works very well in practice (witness low-density parity-check codes and turbo codes).

Decoding a Turbo Code:



(Some) connections to later talks:

- Use "local" cycle-free neighborhood to perform approximate analysis (Richardson, Urbanke).
- Let variables all have degree two; they perform no computation and can be "absorbed" as edges. $\rightarrow$ Generalized state-space representations (Forney graphs).
- Use the structure of the graph to build highly parallel decoders. (analog decoders: Loeliger)

Conclusions:

Factor graphs provide a natural description of the factorization of a global function into a product of local functions.

The sum-product algorithm connects a variety of well-known algorithms in a common framework.